






School of Computing Technologies

COSC1073 Programming 1 2023 S1

Assessment 3

	<p>Assessment Type: Individual assignment; no group work.</p> <p>Submit online via Canvas → Assignments → Assignment 3.</p> <p>Marks awarded for meeting requirements as closely as possible.</p> <p>Clarifications/updates may be made via announcements/relevant discussion forums.</p>
	<p>Due date: 12 June 9:00am</p> <p>Please check Canvas → Assignments → Assignment 3 for the most up to date information.</p> <p>As this is a major final assignment, late penalty of 20% per each working day applies for up to 5 working days late, unless special consideration has been granted.</p>
	<p>Weighting: 30 marks out of 100</p>

Background Information

For this assessment you need to write an object-oriented console application in the Java programming language which adheres to the following basic object-oriented programming principles:

- Your code should follow good object-oriented principles such as encapsulation, composition, cohesion, etc...
- Setting the visibility of all instance variables to private.
- Using getter and setter methods where needed and appropriate with consideration given to scope & visibility.
- Use static variables and methods when there is good reason to do so, such as reducing code repetition whilst still maintaining good object-oriented design.

You are being assessed on your ability to write a program in an object-oriented manner. Writing the program in a procedural style is NOT permitted and will be considered inadmissible receiving a **zero** grade. You will also receive **zero** if your submission does not compile on command line, e.g. by

```
> javac A3_P1_2023.java
```

The following guidelines apply throughout this assessment:

- You **must** use an array to store any collections of objects in your program.
- You must **not** use ArrayList, HashMap or any other data structures within the Java API or from 3rd party libraries.

Overview

You have a start-up. Your first contract is to build a Java application for a local grocery shop. This system is required to have three main functions: (1) uploading item information (2) reading in order information from customers (3) showing the summary accordingly.

You are provided with a template code, with some sample data files provided. To complete the system, you will implement or add several classes. We give introduction of each class and detailed instructions at each level from PASS, to CR, to DI and HD. Sample outputs are also provided below for each level.

===== PASS LEVEL =====

Item class – It has (at least) four instance variables: **String** name, **double** unitCost, **double** unitPrice, **int** stock.

It has only one constructor which takes four parameters, e.g.

```
Item("Apple", 12.7, 20.5, 50)
```

Parameters represent (1) the name of the item, (2) its cost per unit, e.g. in kg, in box, that is the buying price from wholesalers, (3) its selling price per unit, e.g. in kg, in box, (4) the available stock which is initialised at the beginning. In the above example, the item is Apple, buying at \$12.7 per box, selling at \$20.5 per box, with 50 boxes initially added into the stock.

The information of items are stored in "**items.txt**" which is included in the provided zip file. See below for its content. **This shop only sells these 7 items**. You may change the values to test your program. But DO NOT change the item name nor the order.

```
Apple 12.7 20.5 50
Banana 6.55 7.8 30
Coke 1 1.5 100
Donut 2 4 20
Egg 1.15 3.3 50
Fish 13.5 22.2 10
Grapes 1.3 3.5 3
```

You can assume the file will never be corrupted. Its data will always be in the right data format, although the value might be invalid (see later for details).

Order class – Each order has a **customer** which is a String, and an int array **orderedItems** to store the details of the order. There is a **capacity** in the class. It is always **20** as the shop cannot handle orders that contains more than 20 items. The **fee** for handling/delivering one order is always \$9.99, unless it is set by one of its methods **charge(double fee)**.

The order info are stored also in a file. An example "**orders.txt**" is included in the provided zip file. Below is its content. You may change values to test your program. The first line is the total number of orders. Each line is an order starting with the customer name (in one string, no spaces), then 7 ints for the seven kinds of items listed above. The sequence is the same as that in items. A value 0 means item not order. For example Frodo ordered 1 unit of banana, 2 units of coke, 1 unit of egg, 2 units of fish and 1 unit of grapes, no apple and donut.

```
4
Frodo_Baggins 0 1 2 0 1 2 1
Mary_Popping 2 1 3 10 0 0 0
Farhad_Osman 0 0 0 0 10 0 10
Mary_Popping 1 2 3 4 3 2 1
```

You can also assume that the order file is never corrupted. Its data always are in the right data format, although it may contain invalid values (see later for details).

A3_P1_2023 class – This is the driver class that initializes and runs the order management functionalities for the grocery shop. When your program compiles and runs from command line, it should look like what's shown below. No need for your program to output exactly the same, e.g. spacing/alignment, as long as your output is formatted and looks close enough.

```
> javac A3_P1_2003.java
> java A3_P1_2003 orders.txt

===== Welcome to Java Grocery =====
1.      Apple:      12.70      20.50      50
2.      Banana:      6.55      7.80      30
3.      Coke:       1.00      1.50     100
4.      Donut:       2.00      4.00      20
5.      Egg:        1.15      3.30      50
6.      Fish:      13.50     22.20      10
7.      Grapes:      1.30      3.50       3

=====      4 Order(s)      =====
Frodo_Baggins: 0 1 2 0 1 2 1
Mary_Popping: 2 1 3 10 0 0 0
Farhad_Osman: 0 0 0 0 10 0 10
Mary_Popping: 1 2 3 4 3 2 1
```

Two methods `readItems()`, `readOrders(String orderFile)` have already been implemented in the template code. They read the item data file, the order data file and return the item list and order list respectively to the main method.

```
Item[] items = readItems();
Order[] orders = readOrders(args[0]);
```

You may not need to change these two methods at this level. However understanding the implementation of them certainly would be very helpful for the tasks. For higher levels, you may need to update them accordingly to accomplish some of the advanced tasks.

===== CR LEVEL =====

NOTE: You should only attempt this level if you have completed PASS level.

Define your own exception `InvalidItemException`, which will be raised when an item is to be created upon invalid data. The values for the cost, the selling price and the stock must be in the range of 0 to 1000, inclusive. So prices like \$0, \$58.91 and \$1000 per unit are all fine, but not \$-3.0, nor \$1000.1. Stock cannot be negative or more than 1000 either. In addition, the item name cannot be over 10 characters long (assume no missing names). Items that do not meet these requirements will trigger this customised exception during creation. If you make a change to items.txt such as change Apple to AppleRoyalGala, set Donut's stock to 2000, Egg's cost to -5, Grapes' price to 1000.5, your program should output:

```
> java A3_P1_2003 orders.txt
===== Welcome to Java Grocery =====
Check items.txt, remove all invalid item values and restart the program
```

If there is no error in the item file, then your program would still display the item list as that in the PASS level, but now with a more presentable list of orders. Each order will show the name and the ordered quantity for each item on the order. Items with zero, meaning not ordered, will not show. See the examples on the next page. Again, your output should be formatted, but no need to be exactly the same as the example.

```

===== 4 Order(s) =====
# Frodo_Baggins
2.    Banana:  1
3.     Coke:   2
5.     Egg:    1
6.     Fish:   2
7.     Grapes: 1

# Mary_Popping
1.     Apple:   2
2.     Banana:  1
3.     Coke:    3
4.     Donut:  10

# Farhad_Osman
5.     Egg:    10
7.     Grapes: 10

# Mary_Popping
1.     Apple:   1
2.     Banana:  2
3.     Coke:    3
4.     Donut:   4
5.     Egg:     3
6.     Fish:    2
7.     Grapes:  1

```

===== DI LEVEL =====

NOTE: You should only attempt this level if you have completed CR level.

Define another customised exception **InvalidOrderException**, which is raised when creating an order with invalid data. As stated in PASS level, class Order has a capacity 20, the maximum number of items on each order. If an order that has a total more than 20 will trigger an **InvalidOrderException**. Similarly your program would also raise this exception, if the order contains a negative number or the total is zero (a meaningless order).

If there is a problem in the items.txt, your program will show the same error as in the CR Level. If there is no problem in items, but in orders, then your program will output:

```

> java A3_P1_2003 orders.txt
===== Welcome to Java Grocery =====
Check the order file, fix errors of all orders and restart the program

```

If a different exception occurs, your program will display a different message.

```

java A3_P1_2003 orders.txt
===== Welcome to Java Grocery =====
Something else other than items/orders is wrong

```

In addition, at this level, your program output is more informative, showing the unit price, the total price of each item and the total of the order. Moreover, it shows the item list again, assuming all of the orders have been fulfilled. So the stock values on the list will be reduced.

```

===== 4 Order(s) =====
# Frodo_Baggins
2.    Banana:  1      7.80      7.80
3.     Coke:   2      1.50      3.00
5.     Egg:    1      3.30      3.30
6.     Fish:   2     22.20     44.40
7.     Grapes: 10      3.50     35.00
Total: ----- 93.50 -----

```

```
# Mary_Popping
1.   Apple:   2      20.50    41.00
2.   Banana:  1       7.80     7.80
3.    Coke:   3       1.50     4.50
4.   Donut:  10       4.00    40.00
Total: ----- 93.30 -----
```

```
# Farhad_Osman
5.    Egg:   10       3.30    33.00
7.   Grapes: 10       3.50    35.00
Total: ----- 68.00 -----
```

```
# Mary_Popping
1.   Apple:   1      20.50    20.50
2.   Banana:  2       7.80    15.60
3.    Coke:   3       1.50     4.50
4.   Donut:   4       4.00    16.00
5.    Egg:    3       3.30     9.90
6.   Fish:    2      22.20    44.40
7.   Grapes:  1       3.50     3.50
Total: ----- 114.40 -----
```

All orders together >>>> 369.20

```
===== Updated Item List =====
1.   Apple:   12.70    20.50    47
2.   Banana:   6.55     7.80    26
3.    Coke:    1.00     1.50    92
4.   Donut:    2.00     4.00     6
5.    Egg:     1.15     3.30    36
6.   Fish:    13.50    22.20     6
7.   Grapes:   1.30     3.50   -18
```

===== HD LEVEL =====

NOTE: You should only attempt this level if you have completed DI level.

You may have noticed the negative stock of grapes in the above DI example. At this level, you are required to deal with that problem. You need to define another exception, **NotEnoughStockException** in case of a negative stock. If the stock is not enough, that part of the order will not be fulfilled, e.g. no sale. In addition, the fee of each order needs to be considered at this level (not required in DI level), especially with the sub-classes.

There are two special types of orders, **SelfPickupOrder** and **VIPOrder**. The first one is for orders that will be picked up by the customer at the shop. So there is no charge for the order handling. The second type, **VIPOrder**, is for VIP customers. If the total price on the order (excluding the handling fee) is over \$50, then a 15% of discount will apply to the part over 50. So a total of \$70 will get $15\% * (70-50) = \$3$ discount. The fee for a VIP order is always \$5.00.

Note, **SelfPickupOrder** and **VIPOrder** are mutually exclusive, meaning an order cannot in both types. A **VIPOrder** will still charge the \$5 handling fee, even if the customer decides to pick it up in person from the shop.

An order starts with “**P_**” in the order file is a **SelfPickupOrder**. An order starts with “**V_**” in the order file is a **VIPOrder**. Below is another order file “orders2.txt” that is also included in the zip file. The fourth order will be picked up by Mary. The last order is from VIP Obi Wan. Note the total on his order is less than the total of all items on his order because of the discount.

```

5
Frodo_Baggins 0 1 2 0 1 2 10
Mary_Popping 2 1 3 10 0 0 0
Farhad_Osman 0 0 0 0 10 0 10
P_Mary_Popping 1 2 3 4 3 2 1
V_ObiWan_Kenobi 10 10 0 0 0 0 0

```

```

> java A3_P1_2003 orders2.txt
===== Welcome to Java Grocery =====
1.      Apple:      12.70      20.50      50
2.      Banana:      6.55      7.80      30
3.      Coke:       1.00      1.50     100
4.      Donut:       2.00      4.00      20
5.      Egg:        1.15      3.30      50
6.      Fish:       13.50     22.20      10
7.      Grapes:      1.30      3.50       3
=====
5 Order(s) =====
# Frodo_Baggins
2.      Banana:      1        7.80      7.80
3.      Coke:        2        1.50      3.00
5.      Egg:         1        3.30      3.30
6.      Fish:        2       22.20     44.40
7.      Grapes:     10        3.50    ** Not Enough Stock **
Total: ----- 58.50 + 9.99 = 68.49

# Mary_Popping
1.      Apple:       2       20.50     41.00
2.      Banana:      1        7.80      7.80
3.      Coke:        3        1.50      4.50
4.      Donut:     10        4.00     40.00
Total: ----- 93.30 + 9.99 = 103.29

# Farhad_Osman
5.      Egg:       10        3.30     33.00
7.      Grapes:   10        3.50    ** Not Enough Stock **
Total: ----- 33.00 + 9.99 = 42.99

# P_Mary_Popping
1.      Apple:      1       20.50     20.50
2.      Banana:     2        7.80     15.60
3.      Coke:       3        1.50      4.50
4.      Donut:      4        4.00     16.00
5.      Egg:        3        3.30      9.90
6.      Fish:       2       22.20     44.40
7.      Grapes:     1        3.50      3.50
Total: ----- 114.40 + 0.00 = 114.40

# V_ObiWan_Kenobi
1.      Apple:     10       20.50    205.00
2.      Banana:    10        7.80     78.00
Total: ----- 248.05 + 5.00 = 253.05

All orders together >>>> 582.22

===== Updated Item List =====
1.      Apple:      12.70      20.50     37
2.      Banana:      6.55      7.80      16
3.      Coke:       1.00      1.50     92
4.      Donut:       2.00      4.00       6
5.      Egg:        1.15      3.30     36
6.      Fish:       13.50     22.20       6
7.      Grapes:      1.30      3.50       2

```

NOTE: It is expected you will be able to build the system with the information provided. The requirements provided above **must** not be adjusted/modified. However, you may also **add** to the design presented, for example adding supplementary implementation details, such as additional methods / constants, etc...

Marking Guide

Achieving PASS level functionalities	[15 marks]
• Item class	[3 marks]
• Order class	[3 marks]
• Display the item list properly	[4 marks]
• Display the order list properly	[5 marks]
Achieving CR level functionalities	[3 marks]
• Define InvalidItemException	[0.5 marks]
• Exception handling	[1 mark]
• Display the detailed order list	[1.5 marks]
Achieving DI level functionalities	[3 marks]
• Define/Handling of InvalidOrderException and other exceptions	[1 mark]
• Correct display of the unit price and total price for each line	[1 mark]
• Correct display of the total for each order and the updated item list	[1 mark]
Achieving HD level functionalities	[6 marks]
• Define/Handling of NotEnoughStockException	[0.5 marks]
• SelfPickupOrder	[0.5 marks]
• VIPOrder	[1.5 marks]
• Proper use of polymorphism	[1 mark]
• Correct output of the order list according to the HD requirement	[2 marks]
• Correct output of the updated item list at the end	[0.5 marks]
Code quality	[3 marks]
• Variable names; indentation; comments	[1 mark]
• Clear structure, no overly long methods (methods >= 50 lines)	[1 mark]
• Proper OO, good style, no magic numbers	[1 mark]
Total	[30 marks]

NOTE: Non-object-oriented program, or not compilable, or not runnable under command line will be treated as unacceptable and will receive **zero** mark.

Submission Instructions

Submit via zip file

Export the project as an archive; the file name should be the same as the project name, for example **s1234567-A3.zip** and submit the zip file to Canvas under [Assignments](#).

To export a project as a zip file in IntelliJ, click **File -> Export -> Project to zip file**.

NOTE: you need to make sure your code compiles and runs properly on command line.

Academic Integrity and Plagiarism (Standard RMIT Warning)

Your code will be automatically checked for similarity against other submissions so please make sure your submitted project is entirely your own work.

Academic integrity is about honest presentation of your academic work. It means acknowledging the work of others while developing your own insights, knowledge, and ideas. You should take extreme care that you have:

- Acknowledged words, data, diagrams, models, frameworks and / or ideas of others you have quoted (i.e., directly copied), summarised, paraphrased, discussed, or mentioned in your assessment through the appropriate referencing methods.
- Provided a reference list of the publication details so your reader can locate the source if necessary. This includes material taken from internet sites.

If you do not acknowledge the sources of your material, you may be accused of plagiarism because you have passed off the work and ideas of another person without appropriate referencing, as if they were your own.

RMIT University treats plagiarism as a very serious offence constituting misconduct. Plagiarism covers a variety of inappropriate behaviours, including:

- Failure to properly document a source.
- Copyright material from the internet or databases.
- Collusion between students.

For further information on our policies and procedures, please refer to the [University website](#).

Assessment Declaration

When you submit your project electronically, you agree to the RMIT [Assessment Declaration](#).